Enabling Event Tracing at Leadership-Class Scale through I/O Forwarding Middleware

Thomas Ilsche[*‡], Joseph Schuchart[*], Jason Cope[†], Dries Kimpe[†], Terry Jones[‡], Andreas Knüpfer[*], Kamil Iskra[†], Robert Ross[†], Wolfgang E. Nagel[*], Stephen Poole[‡]

[*]Technische Universität Dresden, ZIH 01062 Dresden, Germany
{thomas.ilsche,joseph.schuchart,andreas.knuepfer,wolfgang.nagel}@tu-dresden.de

[†]Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439, USA
{copej,dkimpe,iskra,rross}@mcs.anl.gov

[‡]Oak Ridge National Laboratory, Mailstop 5164, Oak Ridge, TN 37831, USA
{trjones,spoole}@ornl.gov

ABSTRACT

Event tracing is an important tool for understanding the performance of parallel applications. As concurrency increases in leadership-class computing systems, the quantity of performance log data can overload the parallel file system, perturbing the application being observed. In this work we present a solution for event tracing at leadership scales. We enhance the I/O forwarding system software to aggregate and reorganize log data prior to writing to the storage system, significantly reducing the burden on the underlying file system for this type of traffic. Furthermore, we augment the I/O forwarding system with a write buffering capability to limit the impact of artificial perturbations from log data accesses on traced applications. To validate the approach, we modify the Vampir tracing toolset to take advantage of this new capability and show that the approach increases the maximum traced application size by a factor of 5x to more than 200,000 processes.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Tracing*; D.4.3 [**Operating Systems**]: File Systems Management—*File organization*; D.4.4 [**Operating Systems**]: Communications Management—*Input/output*

Keywords

event tracing, I/O forwarding, atomic append

1. INTRODUCTION

Performance analysis tools are a vital part of the HPC software ecosystem. They provide insight into the runtime behavior of parallel applications and guide performance optimization activities toward the most promising or urgent aspects. Porting and tuning performance measurement tools are essential since they must be efficient and must not perturb the runtime behavior of the analyzed application, even at full scale.

The rising levels of concurrency in leadership-class systems present a number of challenges to performance analysis tools. One challenge to scaling these tools is efficiently storing trace data. These tools often generate large amounts of data and must execute efficiently at full scale. Traditional access patterns for these tools, such as file per process, file per thread, and synchronous I/O, do not scale well past tens of thousands of processes. Such data access patterns require excessive use of file metadata operations and overwhelm leadership-class storage systems. Synchronous I/O may cause unnecessary delays in trace data collection and skew application execution. Alternative access patterns, such as a shared file pattern, may alleviate metadata bottlenecks but inject artificial synchronization into the application. Therefore, a unique data organization is desired that exploits the log like I/O behavior of performance analysis tools, allows for uncoordinated access to a shared file from multiple processes, and tolerates lazy I/O semantics.

To achieve and sustain application event trace recording at full leadership-class scale, we investigated several I/O optimizations to support high-performance, scalable, and uncoordinated access of event trace data generated by the Vampir toolset [17]. We observed that the uncoordinated I/O patterns generated by the VampirTrace and Open Trace Format (OTF) tools could be transparently optimized at an intermediate file I/O aggregation layer, known as the I/O forwarding layer. We integrated the I/O Forwarding Scalability Layer (IOFSL) [2, 23] with the VampirTrace/OTF toolset. Also as part of contributions of this paper, we implemented optimizations and new capabilities within IOFSL to reorganize and optimize the captured VampirTrace/OTF I/O patterns while preserving the independent I/O requirements of these tracing tools. These new features include a distributed atomic file append capability and a write buffering capability. By taking advantage of characteristics of the event trace workload and augmenting our HPC I/O stack to better support it, we have reduced the stress that the trace I/O workload places on HPC storage systems. Furthermore, we have reduced the impact of HPC I/O storage systems on the tracing tools.

Our investigation and evaluation resulted in significant improvements to the VampirTrace and OTF software stack. We increased the VampirTrace and OTF tracing infrastructure scalability by 5x (a $40,000$ core to $200,000$ core capability improvement), generated a trace containing 941 billion events at an average aggregate trace data storage rate of 13.3 billion events per second, and demonstrated that coupling IOFSL and the Vampir toolset yields a performance analysis framework suitable for end users on leadership-class computing systems. Overall, we have shown that the entire software stack including trace generation, middleware, post-processing, and analysis can be utilized to analyze a parallel application consisting of 200,448 processes.

The remainder of this paper is organized as follows. The general I/O requirements of performance analysis tools and the Vampir toolset I/O needs are described in Section 2. An overview of IOFSL and optimizations relevant to tracing is provided in Section 3. Section 4 describes the integration and scalability improvement efforts. The proposed concepts are evaluated on a leadership-class machine, and the resulting performance measurements are analyzed in Section 5. An overview of related work is given in Section 6. Conclusions and insights into future work are summarized in Section 7.

2. THE VAMPIR TOOLSET

The Vampir toolset is a sophisticated performance analysis infrastructure for parallel programs that use combinations of MPI, OpenMP, PThreads, CUDA, and OpenCL. It consists of the Vampir GUI for interactive post-mortem visualization, the VampirServer for parallel analysis, the VampirTrace instrumentation and runtime recording system, and the Open Trace Format as the file format and access library. The Vampir toolset relies on event trace recording, which allows the most detailed analysis of the parallel behavior of target applications. First, the Vampir toolset performs instrumentation of the target application using various techniques. During run time, the monitoring component collects the instrumented events together with significant properties. These include entry/exit events for user code subroutines, message send/receive events, collective communication events, shared memory synchronization, and I/O events. A single Vampir event needs approximately 10 to 50 bytes for its encoding in the buffer. Typically, event frequencies range from 100 to $100,000$ per second (with proper settings). A parallel run with $10,000$ processes or threads for 10 minutes results in data sizes of $6 \cdot 10^9$ to $3 \cdot 10^{13}$ bytes (approx. 5.6 GB to 27 TB[1]). The trace buffer size should not exceed the local main memory size minus the memory required by the target application; otherwise the application behavior will be severely distorted. Typical sizes are 10 MB to 1 GB per process or thread. The event trace data is written to a set of OTF files and is then ready for post-mortem investigation with the Vampir GUI. By default, VampirTrace and OTF use a file-per-thread I/O pattern to store data to minimize coordination.

[1]In this paper, we use $1\,\mathrm{MB} = 2^{20}\mathrm{B}$, $1\,\mathrm{GB} = 2^{30}\mathrm{B}$, and $1\,\mathrm{TB} = 2^{40}\mathrm{B}$.

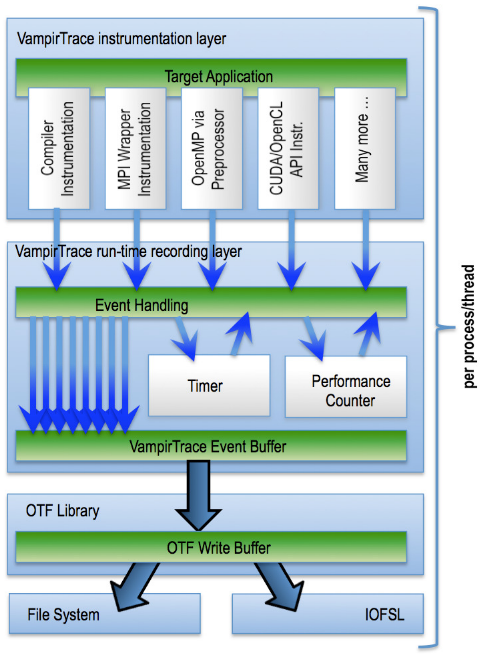


Figure 1: The VampirTrace data flow.

Figure 1 gives an overview of VampirTrace's data flow. Additional information about Vampir, VampirTrace, and OTF is provided in our prior work [17]. When the VampirTrace monitoring component captures the parallel runtime behavior of the target application, it strives to impose minimal perturbation. Triggered by events of interest, the runtime recording layer stores the event information together with vital properties (precise time stamp, event-specific properties, performance counter values if configured) to a preallocated memory buffer. The recording is performed independently for every process or thread into local buffers. In order to avoid artificial synchronization of the target application, buffers are never shared between processes or threads.

2.1 I/O Patterns in VampirTrace

The VampirTrace buffer can be written to an OTF file in several ways. The most general one is to flush the buffer as soon as it is filled. If the measurement library cannot ensure enough space for an event in the buffer, the application is delayed, and the data is written to a file through the OTF library. These buffer flush phases are clearly marked in the trace so that their effect is not mistaken for stray behavior of the target application. However, they can delay other processes waiting for messages or synchronization in the application, unless the buffer limit is reached at the same spot in the application for all processes (i.e., each process generated the same number of events). This becomes an issue for larger scale and tightly coupled applications.

Alternatively, VampirTrace can use collective MPI operations to trigger synchronized buffer flushes by piggybacking work on application collective operations. For each collective operation, the measurement environment communicates the maximum buffer level. Once a threshold is reached, all processes enter a flush phase. All synchronous flushes and synchronizations are captured in the trace and clearly marked for analysis. An additional barrier after the flush makes sure the processes resume simultaneously, avoiding an indirect impact on the application behavior.

Having a single flush at the end of the recording (typically during an `MPI_Finalize` wrapper) is preferred when possible because it removes trace I/O from the application execution. For this case, the event buffer must be able to hold all events generated during the application execution. This can be achieved by reducing the total event count, for example, by using filters or tracing only specific iterations of the application. In addition, transparent compression using zlib in the OTF layer helps reducing the file sizes. It is applied only to the output during a buffer flush to keep the perturbation minimal at the expense of the required memory buffer space.

In general, the number of buffer flushes per process can be limited to avoid uncontrolled use of storage space. The specialized cases for flushing largely improve the perturbation due to I/O, but it cannot be guaranteed that a collective operation triggers a buffer flush before the buffer is full. It is also difficult to choose appropriate settings that do not require buffer flushes. Therefore, VampirTrace may need to fall back to the general uncoordinated buffer flush that prevents loss of data at the cost of a performance impact. Traditional collective I/O optimizations and methods rely on implicit synchronization. This required synchronization distorts the measurement and is therefore not an appropriate solution for VampirTrace's data collection. Also, the possibility of processes or threads being created during runtime makes it impossible to know how many participants there might be for collective I/O at any point in time.

2.2 I/O Challenges and Solutions

The original configuration of VampirTrace imposed two I/O challenges that prevented it from tracing applications running at full-scale on leadership-class systems. First, the often nearly simultaneous buffer flushes of many processes or threads increase I/O bandwidth pressure on the I/O subsystem. This pressure can delay trace data storage and skew the application trace measurements. The storage targets can get overwhelmed with inefficient, nearly random workloads that can sap over 60% of their peak performance. Second, the metadata load for this configuration is high because of the creation of many individual files and the allocation of file system blocks for a large number of I/O operations. For example, ORNL's JaguarPF is in principle capable of opening one file per process even at the scale of $224,000$ processes, taking around 45 seconds (David Dillow, personal communication, September 20, 2011). In production usage however, such operations have been observed to take five minutes [30]. Parallel file creation requests at a high rate will impact all other users and jobs on the machine. VampirTrace/OTF supports the use of node-local storage for the intermediate trace I/O. This involves copying the files from local to global directories after the measurement. However, node-local storage is not available on many current large-scale systems and the situation will not change on the next generation systems (Cray XK6, Blue Gene/Q) either.

To address these challenges, we identified several opportunities that allow VampirTrace to store its trace data collections more efficiently while minimizing the overhead of the data accesses on the traced applications. We found that a shared file access pattern may be better suited for large-scale applications than is the file-per-thread access pattern. This approach significantly reduces the metadata load of the file-per-thread access pattern. Since writing to a shared file from many processes requires coordination among those processes, we opted to store trace data collections in multiple files, where the total number of files is far less than the number of traced processes or threads. For storing the trace data collections, we identified an append-only streaming access pattern that is easier for parallel file systems to handle than are random I/O patterns. To further reduce the impact of file system performance on trace data collection, we recognized that a write buffering strategy can isolate file system performance from the trace data collection storage.

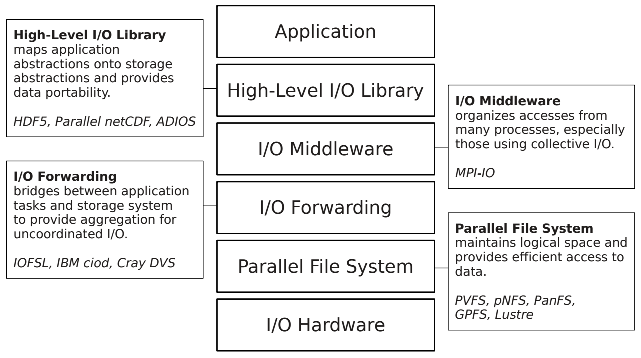
The capabilities required by the improved VampirTrace I/O access pattern are not readily available or adequately supported by vendor-supplied I/O software stacks. The necessary capabilities missing from these stacks include transparent aggregation of uncoordinated I/O requests to a set of files, portable atomic append of file data, and write buffering. These capabilities can be implemented within I/O forwarding tools, such as IOFSL. Our I/O forwarding approach provides a convenient solution that integrates with the existing VampirTrace/OTF infrastructure and promises to scale much farther than today's high-end systems.

3. THE IOFSL I/O FORWARDING LAYER

The goal of I/O forwarding is to bridge computation and storage systems in leadership-class computers. Using this software, all application file I/O requests are shipped to dedicated resources that aggregate and execute the requests. This approach allows I/O forwarding layers to bridge compute nodes, networks, and storage systems that are physically disconnected, such as on the IBM Blue Gene systems [32]. I/O forwarding middleware aggregates file I/O requests from multiple distributed sources (I/O forwarding clients) to a smaller number of I/O handlers (the I/O forwarding servers). The I/O forwarding server delegates and executes the requests on behalf of the clients. Since the I/O forwarding layer has access to all the file I/O requests, one can implement file I/O optimizations on both coordinated and uncoordinated file access patterns. These optimizations include coalescing, merging, transforming, and buffering I/O requests.

IOFSL [2, 23] is a high-performance, portable I/O forwarding layer for leadership-class computing systems. For communication between the IOFSL clients and servers, the Buffered Message Interface (BMI) library [5] is used. BMI supports native access to the SeaStar2+ network used on the Cray XT platforms using the Portals API and to the IBM Blue Gene/P tree network using ZOID [14]. TCP is supported as a general transport. IOFSL provides a stateless I/O application programming interface called ZOIDFS that applications can use to directly communicate with IOFSL servers. It also provides compatibility layers for POSIX and MPI-IO programming interfaces.

IOFSL is an ideal location to prototype and evaluate new or existing HPC I/O capabilities. Figure 2 illustrates a typical HPC software stack on leadership-class systems and where I/O forwarding fits into it. IOFSL interacts directly with the parallel file system. Implementing our enhance-



High-Level I/O Library
maps application abstractions onto storage abstractions and provides data portability.
HDF5, Parallel netCDF, ADIOS
I/O Forwarding
bridges between application tasks and storage system to provide aggregation for uncoordinated I/O.
IOFSL, IBM ciod, Cray DVS
Application
High-Level I/O Library
I/O Middleware
I/O Forwarding
Parallel File System
I/O Hardware
I/O Middleware
organizes accesses from many processes, especially those using collective I/O.
MPI-IO
Parallel File System
maintains logical space and provides efficient access to data.
PVFS, pNFS, PanFS, GPFS, Lustre

Figure 2: Overview of the HPC I/O software stack.

ments we present in this paper required no changes to vendor-supplied systems software; all of our I/O enhancements were implemented within IOFSL. Since IOFSL is positioned just above the file system, our enhancements to IOFSL can affect all applications using the I/O software stack.

In Section 2.2, we identified several file optimizations that can improve the performance and scalability of the trace data generation of VampirTrace. These improvements include a write buffering strategy to quickly offload trace data from the application compute nodes and an atomic file append capability to reduce random I/O workloads to the file system. In this section, we describe how we implemented these capabilities within IOFSL.

The IOFSL write buffering capability provides nonblocking file I/O enhancements to the IOFSL server and client. This capability transforms blocking I/O operations into nonblocking ones while requiring minimal changes to the application and no changes to the ZOIDFS API. It relaxes the data consistency semantics in order to achieve higher I/O throughputs for the application. To implement this, we modified the file write data path in the IOFSL server to signal operation completion to the client before initiating the file write operation. Once the I/O forwarding server receives the client data for a non-blocking operation, the data is buffered within the server and the client I/O operation is completed. The client is then free to release or reuse its transmitted data buffer since the server is now responsible for completing the I/O operation for the client. The I/O request will complete as soon as the server has resources to process the request or when the client forces all pending nonblocking I/O operations to complete using a commit operation. This behavior allows the IOFSL server to transparently manage nonblocking I/O operations initiated by clients.

IOFSL's atomic append capability allows multiple clients to share the same output file without client-side coordination and supports tools that exhibit log like data access patterns. This file append capability is a distributed and atomic I/O operation. We developed several new IOFSL features to support distributed atomic file append operations. IOFSL servers now provide a distributed hash table that is used to track the end-of-file offset for unique files. This data structure provides a fetch and update operation, so that the current file offset can be retrieved and updated. The distributed storage of file handles allows IOFSL to scatter and decentralize the file offset data. We also developed a mechanism for IOFSL servers to communicate with each other. Originally, IOFSL provided a client-server communication model. The server to sever communication capability allows an IOFSL server to query remote IOFSL servers and retrieve the end-of-file offset information. As a consequence, IOFSL clients are not required to contact multiple IOFSL servers to obtain and update file offset information. Additionally, the IOFSL server coalesces multiple atomic append requests to limit the amount of server to server traffic. The atomic append capability can be used by any number of IOFSL servers, including a local server mode when data files are not shared between IOFSL servers.

There are several benefits to our atomic append approach. Clients can append data to a file that is simultaneously being written to by other clients and I/O forwarding servers. IOFSL clients do not require prior knowledge of the end-of-file position and simply need to deliver the data to be written into the file to the server. This capability effectively allows applications to stream data to the IOFSL servers, which manage data placement within a file. The server returns to the client the file offset where the data was written. This capability is similar to MPI shared file pointers and the O_APPEND mode provided by the POSIX I/O API. The novelty of our approach is in supporting a distributed and portable append functionality — O_APPEND does not work in a multi node environment like a parallel computing system. Since this capability is implemented within IOFSL, it can be used on any system where IOFSL can run, regardless of the underlying file system, operating system, or network. For write buffering operations, the IOFSL server can also return the state of completed operations. The client and application can use this information to construct an index of the data accesses within the file and determine the state of pending nonblocking operations.

4. INTEGRATING OTF WITH IOFSL

The OTF layer provides a single integration point between the VampirTrace stack and IOFSL. Since all trace I/O happens in this layer, this permits a portable solution that is usable for other applications based on OTF. We chose to use the ZOIDFS API because it provides additional capabilities that are not present in the IOFSL POSIX translation layers.

The primary integration goal was to reduce the number of files generated by the Vampir tracing infrastructure. Instead of storing n OTF event streams in n files, we aggregate the streams into m files, where $m \ll n$. m can vary based on the tracing configuration. For example, m could be equal to the number of IOFSL servers (one file per IOFSL server) or be smaller (files shared between the servers). Figure 3 illustrates the file aggregation and integration of the used software layers.

To accomplish this, all ZOIDFS write operations use the novel atomic-append feature of IOFSL. This allows arbitrary subsets of event trace streams to share the same file without any coordination on the OTF side. IOFSL ensures that blocks from the same source stay in their original order but makes no guarantees with respect to global ordering; this approach enables additional optimizations.

The coordination of the blocks and their positions in the shared file is done by IOFSL and the results of this activity is reported to OTF. Every OTF stream[2] collects the file

[2]An OTF stream abstracts the events from a single thread or process.

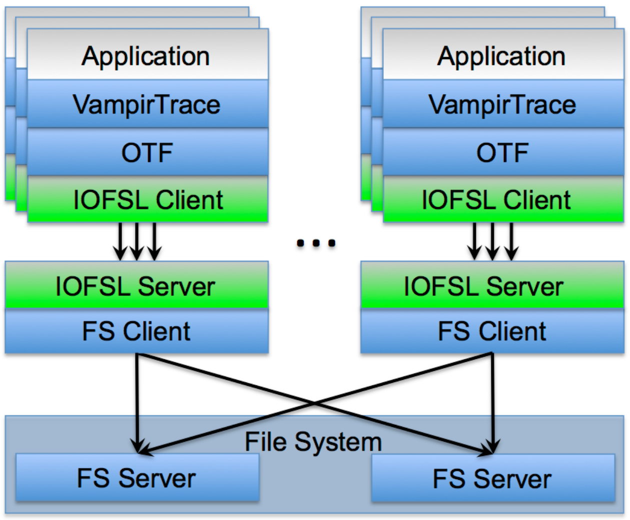


Figure 3: I/O aggregation provided by IOFSL for VampirTrace / OTF. One forwarding server serves multiple clients, usually many servers are used to provide high capacity.

positions (returned by IOFSL) of its blocks individually in memory. As a final new step, all OTF streams write a list of file positions for their own blocks together with their stream identifier. This is sufficient to later extract all blocks from this stream in correct order during reading. The mapping is stored in a shared index file, which is also written via IOFSL using atomic append. The trace data files and index files can be read with or without the use of IOFSL.

The traditional OTF write scheme uses synchronous I/O calls to ensure that all I/O activities happen during the buffer flush phases, which are explicitly marked in the trace. While the flush phases are blocking and application buffers are reused, there can be buffering by the operating system, the standard library, or the file system itself. However, optimizations that just hide the transfer time can negatively affect the application when resources such as I/O bandwidth are used after the flush. The write buffering capability of IOFSL can also decrease the time spent in buffer flushes. Unlike local optimizations however, the trace data has been transferred from the application node to the I/O forwarding node after the completed flush. The file I/O is then initiated from the forwarding node, and no local resources are used after that, minimizing the application perturbation. The effects are similar to other jobs utilizing the shared network and I/O subsystem. In cases where this is undesirable — for example, if the target application's I/O is the subject of the analysis or if the machine's I/O network is not separate from the communication network — the nonblocking I/O capability may not be appropriate and can be disabled. Figure 4 displays the interaction between application, trace library, IOFSL, and the file system in a sequence diagram.

In addition to the improvements necessary to efficiently write the trace output, a number of other optimizations were performed to address scalability bottlenecks within the Vampir toolset. Trace post-processing with `vtunify` was previously parallelized using OpenMP and MPI. The master `vtunify` process serves as a global instance to unify the

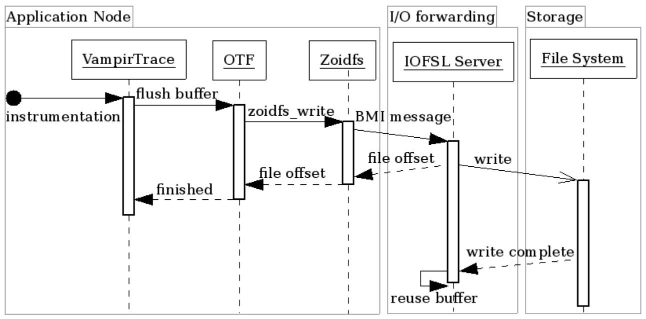


Figure 4: Sequence diagram of a flush that utilizes the buffered I/O capability of IOFSL. Vertical axis is time, increasing downward, not true to scale.

trace definitions (metadata about processes, functions, etc). In order to enable the handling of even larger traces, the serial workload in the master process has been significantly reduced. The remaining serial workload was optimized in time complexity with respect to the total number of application threads. A merge option was implemented in `vtunify`, where each unification worker writes only a single output file instead of one output file per processed stream. This can generate OTF files that are compatible with legacy OTF applications without running into the metadata issues from creating too many files. With these improvements, trace post-processing becomes feasible for large scales, as is documented in Section 5. A hierarchical unification scheme for definitions could further improve the scalability and eliminate the master process as a bottleneck.

As described in Section 2.1, a synchronized flush is beneficial for large-scale tracing scenarios that generate many events. For the required high watermark check, an `MPI_Allreduce` is injected into each global collective operation. At large scales this can result in more significant overhead, since the `MPI_Allreduce` operation is especially prone to high variability caused by operating system noise [16]. In order to reduce the total overhead, a configuration option has been introduced that specifies that the watermark should be checked only every n^{th} collective operation. This mitigates the overhead while still being able to reliably trigger collective synchronized flushes. The additional time used by the measurement library for the watermark check is still clearly marked in the trace file.

We also improved OTF's zlib compression capability. OTF provides zlib with a dedicated compression output buffer. During the OTF and IOFSL integration, the compression capability was updated to ensure that full compression output buffers were written to the file system. This modification ensured that most OTF writes have a fixed size and are stripe aligned, presenting a more efficient pattern to file systems. Unaligned OTF accesses can occur only at the end of the application's execution when the remaining contents of the compression buffer are flushed to the file system.

VampirTrace and the IOFSL integration presented in this paper was designed and tested with hybrid applications that use MPI in combination with OpenMP, threads, CUDA, or other node-local parallel paradigms. No restriction is imposed to when new threads can be created or when buffer flushes may happen.

5. EVALUATION AND ANALYSIS

JaguarPF [4] is a 2.3 petaflop Cray XT5 Supercomputer deployed at the Oak Ridge Leadership Computing Facility (OLCF) at Oak Ridge National Laboratory (ORNL). Data storage is provided by a Lustre-based center wide file system [28].

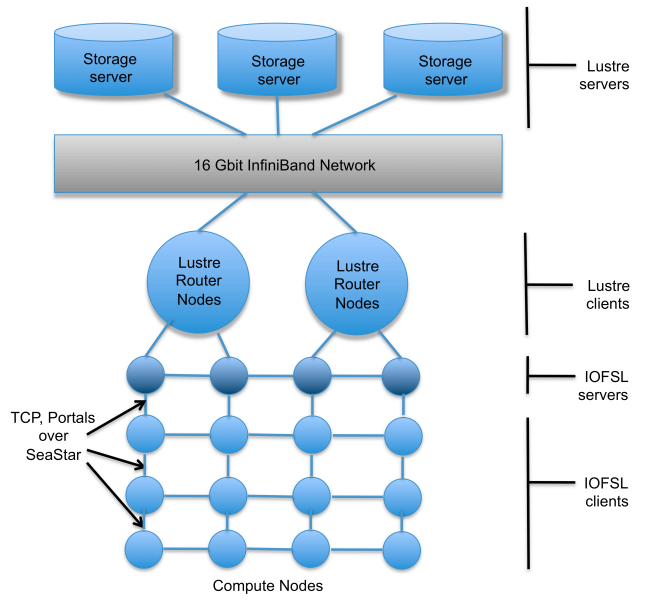


Figure 5: Deployment of application processes and IOFSL servers on JaguarPF.

User-level access to the I/O nodes or Lustre router nodes, which would be optimal locations for running IOFSL servers, is not possible on JaguarPF because of system administration policies. Therefore, we allocate additional compute nodes with each application launch, spawn IOFSL servers on these extra nodes, and proxy all application I/O requests through these nodes. Figure 5 illustrates this deployment strategy. On JaguarPF, the BMI Portals driver is used to leverage the performance of the XT5 SeaStar 2+ interconnect. Using the default IOFSL configuration (unbuffered I/O mode) and the IOR benchmark on JaguarPF when the system was in normal operation, we observed 10.8 GB/s to 11.5 GB/s aggregate sustained bandwidth writing to a single shared file for 1,920 to 192,000 IOFSL clients and when using at most 160 IOFSL servers. We also observed aggregate sustained bandwidths using 60 IOFSL clients per IOFSL server and when writing to unique files (one file per process) of 17.9 GB/s for 2,880 clients, 39.2 GB/s for 5,760 clients, and 42.5 GB/s for 11,520 clients.

To better understand the performance of the new IOFSL's write buffering capability when compared to the original IOFSL synchronous write behavior, we measured the performance of the new capability using a modified version of the IOR benchmark that invoked write buffering operations. These experiments focused on identifying the I/O throughput observed by the application, and the results ignore the cost of application-initiated flushes. Figure 6 illustrates the performance on Oak Ridge's JaguarPF Cray XT5 system. This data clearly indicates that IOFSL can significantly accelerate the sustained storage system bandwidth perceived

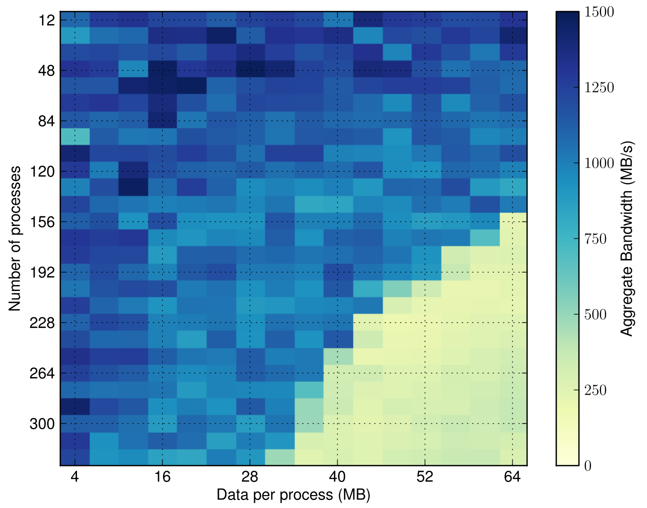


Figure 6: IOFSL write buffering performance on Cray XT5 system (OLCF's JaguarPF). For this experiment, we used a single IOFSL server, 12 to 324 IOFSL clients, and 4 MB to 64 MB of data per IOFSL client.

by the application when sufficient buffer space is available at the IOFSL server. The drop in performance at the bottom right corner of this figure occurs when the IOFSL server exceeds its write buffer space and forces nonblocking I/O operations to act like blocking ones. Therefore, the usefulness of this capability is constrained by the amount of write buffer available to an IOFSL server, the frequency of write buffering operations initiated by IOFSL clients, and the sustained bandwidth the IOFSL server can realize when transferring the buffered data to the storage system.

To demonstrate tracing at large scale, we instrumented the petascale application S3D with VampirTrace. S3D is a parallel direct numerical simulation code developed at Sandia National Laboratories [6]. We used a problem set that scales to the full JaguarPF system. It uses weak scaling to allow a wide range of process counts, from 768 to 200,448. In its role as a early petascale code, S3D is well understood and has been analyzed with TAU and Vampir at lower scales [15]. The purpose of our experiment was to investigate the scaling of trace recording rather than an analysis of the application. S3D provides a real-world instrumentation target for the measurement environment. In addition, the large number of MPI messages generated by S3D creates a high frequency of events (approximately 7,700 events per second per process). We have traced 60 application time steps during our experiments using a basic online function filter. Further improvements of the instrumentation, such as selective function instrumentation or manually tracing a limited number of time steps, were deliberately not applied in order to demonstrate a challenging situation for the measurement environment. The synchronous flush feature in VampirTrace was used with a total of three flushes during the application execution in addition to the final flush during the application shutdown.

Prior to our successful demonstration, the largest scale trace for VampirTrace was approximately $40,000$ processes using POSIX I/O. In practice, achieving this level of parallelism is already difficult because of substantial overhead during file generation and the impact on other users of the file system.

In our demonstration we utilized the full stack that is involved in trace generation: application (S3D), VampirTrace, OTF, IOFSL, the BMI Portals driver for network transfers, and Lustre as a target file system. We have conducted multiple experiments tracing up to 200,448 application cores running S3D and using a set of 672 I/O forwarding nodes resulting in 2,688 files. The largest generated trace size was 4.2 TB of compressed data containing 941 billion events. The total time spent on trace I/O, including forwarding server connection setup, file creation, open, sync, and close, was 71 seconds with write buffering I/O, for a total application run time of 22 minutes. Trace I/O was synchronized among the MPI processes so this time includes the time spent in barriers when waiting for other processes to finish their writing. It therefore represents the total extension of application run time due to trace I/O. On average it took 5.5 seconds for each process to establish the connection to the forwarding server and to open the four shared output files (definitions file and events file, plus an index file for each). Some processes were delayed by up to 32 seconds because of the massive stress on I/O forwarding nodes resulting from write operations from other processes. The intermediate buffer flushes are not affected by connection initialization, file open times, and final commit and therefore show much better individual performance. With write buffering enabled, aggregated write rates of up to 154 GB/s or 33.5 billion events per second were observed, as recorded by the tracing measurement environment during individual flushes. We observed this high bandwidth because all trace data fit into the IOFSL servers' buffers. The cost for the client to flush this data was limited by the IOFSL server performance. This bandwidth result also includes the synchronization of all processes as well as the overhead of OTF and compression.

For comparison, we ran a full-scale experiment using the IOFSL enhancements and unbuffered I/O. The total trace I/O time was 122 seconds, yielding a sustained aggregate bandwidth of 35.3 GB/s. This further indicates that write buffering reduces the I/O overhead observed by the tracing infrastructure. The IOFSL capability buffers trace data at the IOFSL server and overlaps application tracing with trace data storage. In this test series, the trace size per process remains almost constant.

The post-processing (`vtunify`) for such a trace requires approximately 27 minutes but only a fraction of the resources of the application (10,752 workers). This is a required step, regardless of the use of IOFSL. In the post-processing, IOFSL was not utilized since only 10,754 files[3] are created in this step.

This large-scale demonstration shows that trace recording on full-scale leadership-class systems can be done with a well-manageable overhead, even with trace I/O phases during the application execution. We investigated the scaling behavior of our solution with a series of experiments in different configurations. Figure 7 shows the total application run times at different scales with and without tracing. While the overhead of both trace I/O and tracing in general increases

[3]10,752 event files, one definitions file, and one control file.

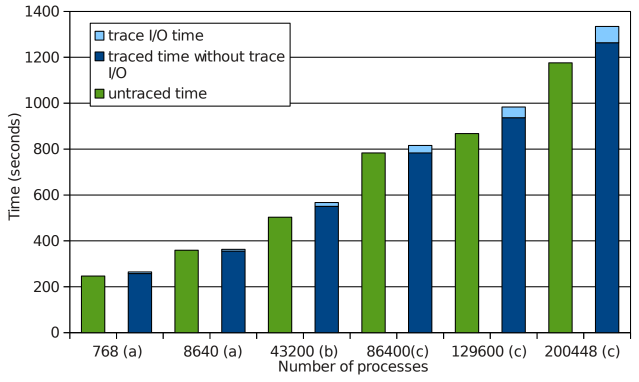


Figure 7: Run times of S3D with and without tracing for different process counts: (a) average of 11 experiments, (b) average of 7 experiments, (c) single test run during dedicated reservation.

with the number of processors, it remains below 15% even at full scale.

A comparison with POSIX I/O at different scales is shown in Figures 8 and 9. The POSIX I/O experiment with 86,400 cores was conducted during a dedicated system reservation. To avoid any potential impact to file system stability, we did not scale the POSIX I/O tests further. For all tests, the same software versions were used — this means that also POSIX I/O tests benefit from those improvements described in Section 4 that are not directly related to IOFSL. The event rate for IOFSL is limited mainly by the I/O throughput of the forwarding servers to the file system, while POSIX I/O is limited by the rate of file creation. The impact of file creation depends on total time, which in turn depends on trace size per process; it will be even more dominant with lower numbers of events.

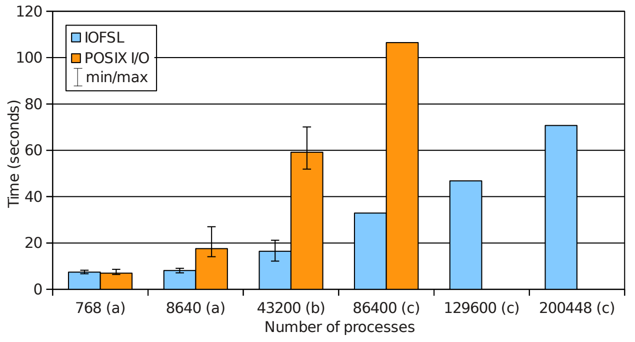


Figure 8: Total trace I/O times for different process counts: (a) average of 11 experiments with min/max, (b) average of 7 experiments with min/max, (c) single test run during dedicated reservation; no POSIX I/O data for 129600, 200448.

The experiments with lower process counts were repeated at different times during production use of the system. I/O in a shared system is always prone to variability, especially with a single metadata server being the bottleneck for any file metadata operation.

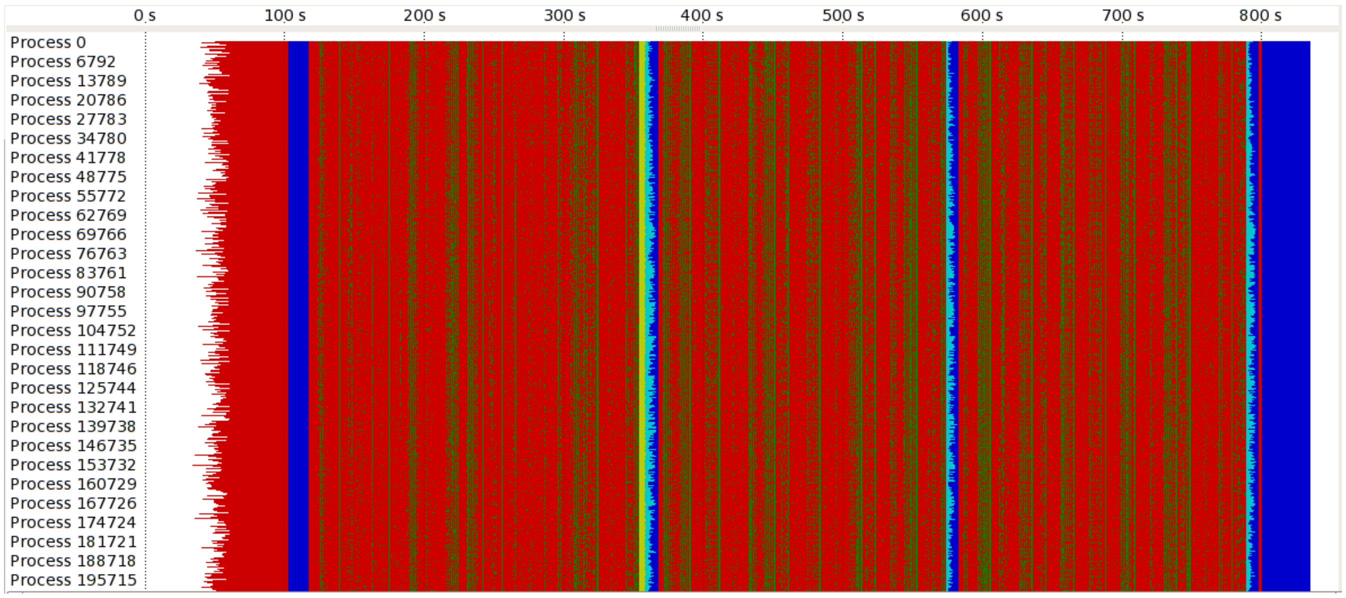


Figure 10: Screenshot of Vampir visualizing a trace of the S3D application using 200,448 cores on JaguarPF. User functions are shown in green, MPI operations in red, and activities of the measurement environment in yellow (file open), light blue (trace I/O) and dark blue (synchronization).

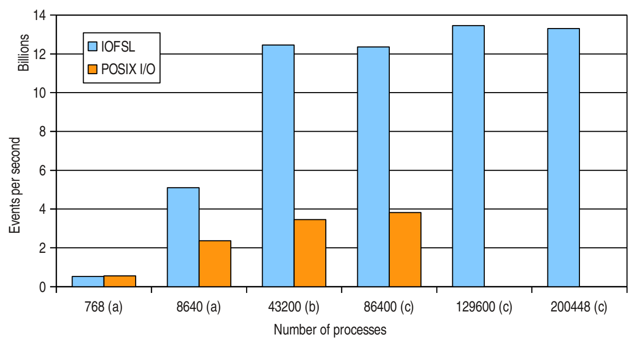


Figure 9: Aggregate event write rates for different process counts. Averages used as in Figure 7. The data includes the overheads of establishing the connection, file open, synchronization, commit, and close.

The trace files generated with IOFSL were validated by using the post-mortem analysis tool Vampir. Vampir was able to read valid trace files and display detailed graphics of measured events versus a timeline and various other displays. Figure 10 shows a trace of S3D with 200,448 processes opened in Vampir, using 21,516 processes for interactive visualization. All processes are visible in an overview showing user functions in green, MPI operations in red, and phases in which the application was suspended by the measurement environment in blue. In addition, file open operations in the first flush are presented in yellow, followed by the actual trace I/O colored light blue as in subsequent flushes. Dark blue represents synchronization phases at the end of each flush. The three flushes take place around 360 s, 580 s, and 790 s after the measurement start. Although being clearly visible, they are sufficiently short in relation to the overall run time of the application. Additionally, the figure shows VampirTrace's internal time synchronizations that extend the first and the last synchronization phase after 100 s and 800 s. This total overview serves as a starting point to further investigate the details by zooming into both the time and process axes.

At such large scales, visualization for analysis purposes becomes more challenging as the ratio between available pixels and displayed processes decreases. Vampir's ability to smoothly zoom and scroll into both the process and time dimension helps to navigate even in such large traces. However, new ways to highlight performance anomalies are required to help the user at those scales find the right spots to focus at. These topics are the subject of ongoing research; our solution lays a foundation for a comprehensive analysis at full scale by providing a feasible way to store event trace data.

We have also evaluated the integrated IOFSL and Vampir toolset on the Intrepid IBM Blue Gene/P (BG/P) leadership-class computing system deployed at the Argonne Leadership Computing Facility (ALCF). The purpose of this evaluation was to demonstrate the portability of our solution to other leadership-class computing platforms, runtime environments, storage systems, and applications.

Intrepid is a 557 teraflop IBM BG/P. It consists of 40,960 compute nodes and 640 I/O nodes, each node with one quad-core 850 MHz PowerPC 450 processor. Intrepid's compute nodes are interconnected by a torus network for point-to-point communication and a tree network for optimized collective operations. The tree network is also used for file I/O; each set of 64 compute nodes is connected over the tree to a dedicated I/O node. I/O nodes communicate via 10 Gigabit Ethernet with one another and with Intrepid's two high-performance storage systems: a 3 PB GPFS file system and a 512 TB PVFS file system.

Intrepid's system administration policies permit users to customize the runtime environment of the system. For our evaluation of IOFSL and VampirTrace/OTF, these policies allowed us to deploy the IOFSL servers on the I/O nodes. To do so, we had to replace IBM's ciod I/O forwarding software with the ZOID [14] BG/P tree network driver to facilitate high-throughput and low-latency communication between user-space processes on the compute and I/O nodes, and boot ZeptoOS [31] operating system on the BG/P compute nodes (replacing IBM's CNK). Additional information on how IOFSL is deployed on BG/P systems is provided in our prior work [23].

We successfully traced the Radix-k [26] image compositing algorithm on Intrepid at a variety of scales using the integrated IOFSL and VampirTrace/OTF toolset. Aside from small adjustments to the infrastructure deployment, the software stack required no additional changes to run on the system. Our evaluation on Intrepid demonstrates that we can trace additional applications, run our toolsets in different runtime environments and systems, and interact with different storage systems. Since our initial target platform was JaguarPF, assessing the performance and scalability of these tools on IBM BG/P systems is a work in progress.

6. RELATED WORK

The performance analysis toolset Scalasca faced similar problems to ours when handling large numbers of trace files. Recently, the scalability of Scalasca was improved up to $300,000$ cores [30]. For tests on a large IBM BG/P system, the SIONlib library was used. It uses a special multifile format that contains additional metadata managing chunks of data from different processes within one file [9]. With SIONlib, multifile creation is a collective operation. This would pose a significant limitation to VampirTrace with respect to the dynamic threading model.

The POSIX I/O standard was designed before the advent of wide-scale parallelism. As such, it suffers from many fundamental characteristics that preclude it from scenarios such as multiple writers updating the same file — a common need for parallel I/O oriented activity [13].

New I/O research efforts within standards-oriented activities have recognized this fact and are actively working on APIs appropriate for extreme-scale parallelism [13,25]. One such API is pNFS [12], an extension to NFSv4 designed to overcome NFS scalability and performance barriers.

MPI-IO [19] provides a more sophisticated I/O abstraction than POSIX. It includes collective operations and file views, which enable coordinated and concurrent access without locking [7]. It does not directly provide an "n-to-m" mapping from clients to output files. OTF's management of mixed blocks in shared files would be difficult to implement on top of MPI-IO, because most implementations (including the popular ROMIO) do not return accurate current file sizes unless a synchronizing collective is used.

The I/O Delegate Cache System (IODC) [22] is a caching mechanism for MPI-IO that resolves cache coherence issues and alleviates the lock contention of I/O servers. IOFSL offers similar capabilities but is located below MPI-IO in the I/O software stack, providing a dedicated abstract device driver enabling unmodified applications to take full advantage of its optimizations.

The I/O forwarding concept was introduced within the Sandia Cplant project [24], which used a forwarding framework based on an extended NFS protocol. IOFSL extends the target environment imagined by Cplant to much larger scales and higher performance through a more sophisticated protocol permitting additional optimizations.

Decoupled and Asynchronous Remote Transfers (DART) [8] and DataStager [1] achieve high-performance transfers on Cray XT5 using dedicated data staging nodes. Unlike our approach, which is transparent to the applications that use POSIX and MPI-IO interfaces, DART requires applications to use a custom API.

Similarly, Adaptable I/O System (ADIOS) [18] provides performance improvements through strategies such as prefetch and write-behind, based on application-specific configuration files read at startup; this information also helps ADIOS minimize the memory footprint during the course of the application run. In contrast, our approach requires no knowledge of the application behavior in advance, and it is situated at a lower level in the I/O software stack.

PLFS [3] is a file system translation layer developed for HPC environments to alleviate scaling problems associated with large numbers of clients writing to a single file. Like our solution, they interpose middleware between the client application and the underlying file system through the use of FUSE. Their solution, which is aimed at checkpointing and similar activities for architectures such as Los Alamos National Laboratory's Roadrunner (3,060 nodes), transparently creates a container structure consisting of subdirectories for each writer as well as index information and other metadata for each corresponding data file. Since our solution is focused on supporting hundreds of thousands of clients or more, we have chosen to aggregate I/O operations in the middleware, thus resulting in fewer metadata operations in the underlying parallel file system. Furthermore, our IOFSL-based solution focuses on transforming uncoordinated file accesses to many unique files, such as a file per process or thread I/O pattern, into a shared file per group of processes I/O pattern. Our solution reduces file system resource contention generated by shared file access patterns (such as file stripe lock contention or false sharing) and eliminates file system metadata overheads generated by I/O patterns with one file per process or thread (such as frequent file creation, stat, or attribute access operations) at extreme scales.

IOFSL work extends the earlier ZOID efforts [14]. ZOID is a Blue Gene-specific function call forwarding infrastructure that is part of the ZeptoOS project. The I/O forwarding protocol used by IOFSL was first prototyped in ZeptoOS. IOFSL is a mature, portable implementation that integrates with common HPC file systems and also works on the Cray XT series and Linux clusters.

While recent work has addressed the use of non-blocking I/O at the I/O forwarding layer [29], our work focuses on providing a portable and transparent to applications, write-buffering based, and high-performance non-blocking I/O capability in HPC environments. Furthermore, non-blocking file I/O capabilities are not provided by existing I/O forwarding tools, including IBM's ciod or Cray's DVS.

In other areas of computer science research, augmentations to existing I/O software that take advantage of specific workload characteristics have been shown effective in improving performance for important workloads. For example, in the Internet services domain, the Google File System provides specialized append operations that allow many tasks to contribute to an output file in an uncoordinated manner [10].

7. CONCLUSIONS AND FUTURE WORK

This paper described the use of I/O forwarding middleware for scalable event trace recording. Through an integration into the OTF library, the Vampir toolset benefits from a new atomic append capability that is provided by the IOFSL I/O forwarding layer. Using the Vampir tracing infrastructure, we demonstrated that this solution enables software tracing at full-scale on leadership-class systems (200,448 processes). A comprehensive trace-based analysis is now feasible for pattern recognition, post-processing, and visualization systems. Within the context of this paper, we have adressed the increasing trace data volumes at large scales at I/O level. Further ongoing work investigates advanced filtering, selective tracing and semantic runtime compression to provide additional benefits for tracing large application runs. We show that even at medium scales, tracing overhead can be significantly reduced with our solution. The benefit for scalability results from reducing the massive amount of metadata file system requests from all application processes to a much lower number. Further improvements on performance comes from utilizing write buffering, which, thanks to being implemented on separate I/O forwarding nodes, does not perturb the application processes. We improved the scalability of the Vampir toolset to leverage the entire performance analysis workflow.

While these results meet our immediate needs and objectives, this effort has led us to consider further related lines of inquiry. We will pursue more advanced aggregate memory footprint optimizations to yield more available memory to user applications. While we have addressed the data collection challenges in this paper and presented a solution to this problem, we do not address how to effectively visualize trace data for applications running at extreme scales. This information visualization challenge will be addressed as our work progresses. We plan to couple the data collection tools and techniques presented in this paper with recent MPI and I/O visualization tools that focus on extreme scale event and trace data collections [20, 21].

The capabilities described in this paper are also applicable to other use cases beyond improving VampirTrace's I/O and can be implemented within other I/O forwarding tools. The new IOFSL capabilities can improve the I/O performance of tools that generate per-process logs. Thus, these capabilities are applicable to massively parallel applications that exhibit log like data storage patterns (such as Qbox's [11] shared file pointer object capability), data-intensive stream processing tools (such as LOFAR's real-time signal processing pipeline [27]), and high-level I/O libraries that allow unlimited dimensionality or enlargement of variable data structures (such as chunked data storage in HDF5). While we limited our demonstration of these capabilities to IOFSL, the capabilities are sufficiently generic and can be implemented within other production-quality I/O forwarding layers, such as IBM's ciod and Cray's DVS. If these capabilities were implemented within these production tools, they could have a substantial impact on the HPC community's ability to understand applications running on leadership-class systems.

The OLCF and ALCF are in the process of upgrading their leadership-class computing resources. The new Titan Cray XK6 supercomputer at OLCF will consist of 299,008 CPU cores and 18,688 GPUs, whereas Mira, a 800,000 CPU core IBM Blue Gene/Q system, will be deployed at ALCF. Both centers will upgrade the storage systems that serve their leadership-class computing resources. While we are confident our toolsets will scale on these systems, we will re-evaluate the scalability and performance of our tools on these new platforms as they are deployed. Moreover, we plan to further investigate the IOFSL and OTF/VampirTrace configuration space on these systems so that we can identify optimal infrastructure configurations for performance analysis I/O workloads.

Acknowledgments

We thank Ramanan Sankaran (ORNL) for providing a working version of S3D as well as a benchmark problem set for JaguarPF. We are very grateful to Matthias Jurenz for his assistance on VampirTrace as well as Matthias Weber and Ronald Geisler for their support for Vampir. The IOFSL project is supported by the DOE Office of Science and National Nuclear Security Administration (NNSA). This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory and the Oak Ridge Leadership Computing Facility at Oak Ridge National Laboratory, which are supported by the Office of Science of the U.S. Department of Energy under contracts DE-AC02-06CH11357 and DE-AC05-00OR22725, respectively.

The general enhancement of the VampirTrace and Vampir tools at TU Dresden for full-size runs on leadership-class HPC systems is supported with funding and cooperation by ORNL and UT-Battelle.

8. REFERENCES

[1] Abbasi, H., Wolf, M., Eisenhauer, G., Klasky, S., Schwan, K., and Zheng, F. DataStager: Scalable data staging services for petascale applications. In *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing (HPDC)* (2009), pp. 39–48.

[2] Ali, N., Carns, P., Iskra, K., Kimpe, D., Lang, S., Latham, R., Ross, R., Ward, L., and Sadayappan, P. Scalable I/O forwarding framework for high-performance computing systems. In *Proceedings of the 11th IEEE International Conference on Cluster Computing (CLUSTER)* (2009).

[3] Bent, J., Gibson, G., Grider, G., McClelland, B., Nowoczynski, P., Nunez, J., Polte, M., and Wingate, M. PLFS: A checkpoint filesystem for parallel applications. In *Proceedings of 21st ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2009).

[4] Bland, A., Kendall, R., Kothe, D., Rogers, J., and Shipman, G. Jaguar: The world's most powerful computer. In *Proceedings of the 51st Cray User Group Meeting (CUG)* (2009).

[5] Carns, P., Ligon III, W., Ross, R., and Wyckoff, P. BMI: A network abstraction layer for parallel I/O. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium, Workshop on Communication Architecture for Clusters (CAC)* (2005).

[6] Chen, J. H., Choudhary, A., de Supinski, B., DeVries, M., Hawkes, E. R., Klasky, S., Liao, W. K., Ma, K. L., Mellor-Crummey, J., Podhorszki, N., Sankaran, R., Shende, S., and Yoo, C. S. Terascale direct numerical simulations of turbulent combustion using S3D. *Computational Science & Discovery 2*, 1 (2009), 015001.

[7] Ching, A., Choudhary, A., Coloma, K., Liao, W., Ross, R., and Gropp, W. Noncontiguous I/O access through MPI-IO. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)* (2003), pp. 104–111.

[8] Docan, C., Parashar, M., and Klasky, S. DART: A substrate for high speed asynchronous data IO. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing (HPDC)* (2008).

[9] Frings, W., Wolf, F., and Petkov, V. Scalable massively parallel I/O to task-local files. In *Proceedings of 21st ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2009).

[10] Ghemawat, S., Gobioff, H., and Leung, S. The Google File System. *SIGOPS Operating Systems Review 37* (Oct. 2003), 29–43.

[11] Gygi, F., Duchemin, I., Donadio, D., and Galli, G. Practical algorithms to facilitate large-scale first-principles molecular dynamics. *Journal of Physics: Conference Series 180*, 1 (2009).

[12] Hildebrand, D., and Honeyman, P. Exporting storage systems in a scalable manner with pNFS. In *Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST)* (2005), pp. 18–27.

[13] IEEE POSIX Standard 1003.1 2004 Edition. `http://www.opengroup.org/onlinepubs/000095399/functions/write.html`.

[14] Iskra, K., Romein, J. W., Yoshii, K., and Beckman, P. ZOID: I/O-forwarding infrastructure for petascale architectures. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)* (2008), pp. 153–162.

[15] Jagode, H., Dongarra, J., Alam, S., Vetter, J., Spear, W., and Malony, A. D. A holistic approach for performance measurement and analysis for petascale applications. In *Proceedings of the 9th International Conference on Computational Science (ICCS)* (2009), vol. 2, pp. 686–695.

[16] Jones, T., Dawson, S., Neely, R., Tuel, W., Brenner, L., Fier, J., Blackmore, R., Caffrey, P., and Maskell, B. Improving the scalability of parallel jobs by adding parallel awareness. In *Proceedings of the 15th ACM/IEEE International Conference on High Performance Networking and Computing (SC)* (2003).

[17] Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M. S., and Nagel, W. E. The Vampir performance analysis tool-set. In *Tools for High Performance Computing* (2008), M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, Eds., Springer Verlag, pp. 139–155.

[18] Lofstead, J. F., Klasky, S., Schwan, K., Podhorszki, N., and Jin, C. Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In *Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments (CLADE)* (2008), pp. 15–24.

[19] MPI Forum. MPI-2: Extensions to the Message-Passing Interface. `http://www.mpi-forum.org/docs/docs.html`, 1997.

[20] Muelder, C., Gygi, F., and Ma, K.-L. Visual analysis of inter-process communication for large-scale parallel computing. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 1129–1136.

[21] Muelder, C., Sigovan, C., Ma, K.-L., Cope, J., Lang, S., Iskra, K., Beckman, P., and Ross, R. Visual analysis of I/O system behavior for high-end computing. In *Proceedings of the 3rd International Workshop on Large-Scale System and Application Performance (LSAP)* (2011).

[22] Nisar, A., Liao, W., and Choudhary, A. Scaling parallel I/O performance through I/O delegate and caching system. In *Proceedings of 20th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2008).

[23] Ohta, K., Kimpe, D., Cope, J., Iskra, K., Ross, R., and Ishikawa, Y. Optimization techniques at the I/O forwarding layer. In *Proceedings of the 12th IEEE International Conference on Cluster Computing (CLUSTER)* (2010).

[24] Pedretti, K., Brightwell, R., and Williams, J. Cplant^{TM} runtime system support for multi-processor and heterogeneous compute nodes. In *Proceedings of the 4th IEEE International Conference on Cluster Computing (CLUSTER)* (2002), pp. 207–214.

[25] Petascale Data Storage Institute. `http://www.pdsi-scidac.org/`.

[26] Peterka, T., Goodell, D., Ross, R., Shen, H.-W., and Thakur, R. A configurable algorithm for parallel image-compositing applications. In *Proceedings of 21st ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2009).

[27] Romein, J. Fcnp: Fast I/O on the Blue Gene/P. In *Parallel and Distributed Processing Techniques and Applications (PDPTA'09)* (2009).

[28] Shipman, G., Dillow, D., Oral, S., and Wang, F. The Spider center wide file system; from concept to reality. In *Proceedings of the 51st Cray User Group Meeting (CUG)* (2009).

[29] Vishwanath, V., Hereld, M., Iskra, K., Kimpe, D., Morozov, V., Papka, M., Ross, R., and Yoshii, K. Accelerating I/O forwarding in IBM Blue Gene/P systems. In *Proceedings of 22nd ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2010).

[30] Wylie, B. J. N., Geimer, M., Mohr, B., Böhme, D., Szebenyi, Z., and Wolf, F. Large-scale performance analysis of Sweep3D with the Scalasca toolset. *Parallel Processing Letters 20*, 4 (2010), 397–414.

[31] Yoshii, K., Iskra, K., Naik, H., Beckman, P., and Broekema, P. C. Performance and scalability evaluation of 'Big Memory' on Blue Gene Linux. *International Journal of High Performance Computing Applications 25*, 2 (2011), 148–160.

[32] Yu, H., Sahoo, R. K., Howson, C., Almási, G., Castaños, J. G., Gupta, M., Moreira, J. E., Parker, J. J., Engelsiepen, T. E., Ross, R. B., Thakur, R., Latham, R., and Gropp, W. D. High performance file I/O for the Blue Gene/L supercomputer. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA)* (2006), pp. 187–196.